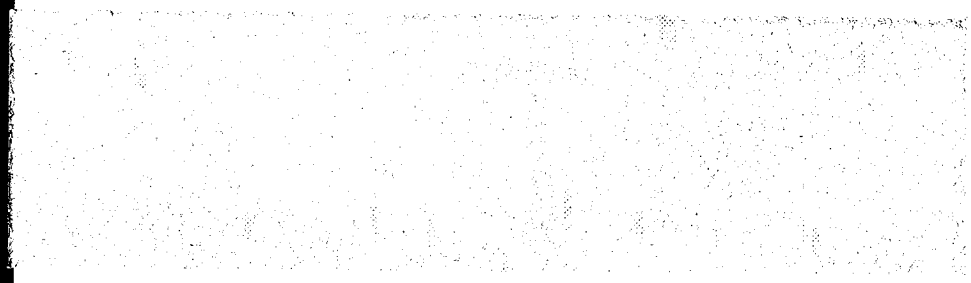

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035

ARC 275 (Rev Feb 81)

Distributed Library

Michael J. Yamasaki

RNR Technical Report RNR-90-008, April 1990

Numerical Aerodynamic Simulation Systems Division
NASA Ames Research Center, Mail Stop T045-1
Moffett Field, California 94035-1000
yamo@nas.nasa.gov

April 16, 1990

Distributed Library

Michael J. Yamasaki

Numerical Aerodynamic Simulation Systems Division
NASA Ames Research Center, Mail Stop T045-1
Moffett Field, California 94035-1000
yamo@nas.nasa.gov

April 16, 1990

Abstract

Distributed Library (Dlib) uses a connection-based, stateful, remote procedure mechanism to provide an environment for the distribution of computation over a heterogeneous network of processors.

1. Introduction

In computational environments which consist of a heterogeneous network of computer systems with a wide range of capabilities, it is often desirable to develop applications which utilize the combined capabilities of more than one system. Distributed Library (dlib) is a tool for developing such distributed applications. The intent of dlib is to allow the development of collections of procedures which work within the context of a remote environment.

Dlib was developed to facilitate the utilization of the computational resources of the Numerical Aerodynamic Simulation (NAS) Processing System Network (NPSN) at NASA Ames Research Center. The NPSN contains a wide range of computer systems, including two high-speed processors (currently, a Cray 2 4/256 and a Cray YMP 8/128) and a small army of Silicon Graphics Iris 4D graphics workstations. Several networks are employed to provide connectivity and a basis for network development and research. These networks include Ethernet, HYPERchannel, ULTRAnet, and Pronet-80.

One use of dlib is to create applications that provide for the interactive visualization of computational fluid dynamics (CFD) data, where the computationally intensive data manipulation occurs on a supercomputer and the rendering and display occurs on a high-performance interactive graphics system. Dlib is designed as a tool to integrate the capabilities of these disparate systems under the control of a single application.

2. Related Work

Most widely used methods for the provision of computational distribution services are based on a remote procedure call (RPC) model. The RPC model and a design are described in the seminal Birrell and Nelson paper [2]. The mechanism for transfer of control and data used in procedure calls is extended in the RPC model to provide for transfer of control and data across a communication network. Birrell and Nelson describe a RPC system as being divided into five parts: the user, the user-stub, the RPC communications package, the server-stub and the server.

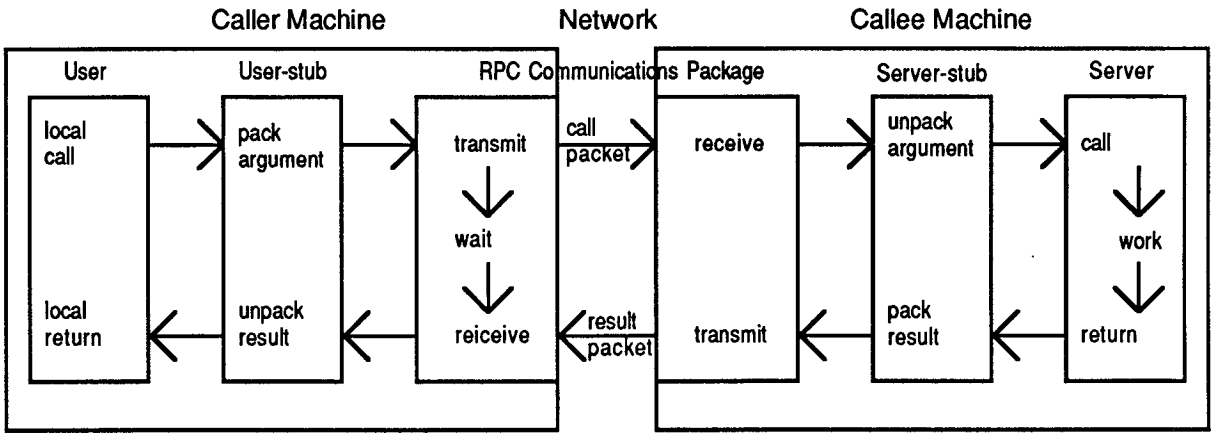

Figure 1. Components and interactions of a simple RPC (after Birrell and Nelson [2])

Some of the design considerations which are implicit in the Birrell and Nelson design are:

i. a system in which the server is at least capable of responding to hundreds of clients

ii. a system in which a client may request service from an array of servers

iii. a system in which response time is constrained by network and process overhead

To this end a distributed database is employed to provide RPC binding services, the server is designed to be as lightweight as possible in order to minimize the elapsed real time between call initiation and response, and a RPC specific transport protocol is used.

The choice of a transport protocol is predicated on the design consideration of a server being capable of responding to a large number of clients and a client being able to request service from a large number of servers. The

relationship between the client and the server is therefore necessarily short and quickly severed, largely persisting for only a single transaction. The overhead associated with connection maintenance and reliable delivery is deemed too high for such a relationship. Consequently, a datagram or unreliable network transport service is often employed as the underlying communication protocol and the higher level RPC protocol implements the required degree of reliability.

The choice of how processes are used in the context of RPCs is similarly based on a transitory relationship between server and client. In some designs the server processes are maintained in an idle state and are activated to handle incoming calls. This allows for calls to be handled without process creation overhead. The short-lived relationship between client and server is preserved by not allowing persistent state information to be stored in the server. The overhead associated with process creation and maintenance of state which is persistent from call to call is also deemed too high for RPCs.

These design choices are duplicated in the design of Sun's RPC [7] and Apollo's NCA/RPC [4]. Although both designs are transport-independant, they are designed for, and work best with, an unreliable transport protocol such as UDP, which is consistent with the Birrell and Nelson design .

Dlib diverges from the notion of a short-lived relationship between client and server. Dlib was developed to provide a service which allows a conversation of arbitrary length within a single context between client and server. It is expected that this relationship between client and server will be long-lived, approaching the duration of the application itself. The dlib server process is designed to be capable of storing state information which persists from call to call, as well as allocating memory for data storage and manipulation.

RPC protocols are likened to local procedure calls without side effects. Dlib more closely resembles the extension of the process environment to include a remote server process. The server process has its own memory segments and process environment. Variables global to the server process may be declared and used as hidden arguments to remote dlib routines. The server process is associated with a single client unless extraordinary provisions are made to respond to more than one client. Special routines are available for the client to share the information stored in server memory segments.

In essence, instead of a remote procedure call service, dlib provides a remote process environment.

3. Distributed Library Overview

The basic steps involved in the use of dlib are:

i. establishment of a connection between the client and the server dispatcher

ii. user authentication

iii. server process creation and rendezvous with client

iv. remote library routine execution

v. server termination

The basic architecture of dlib is illustrated in the following figure:

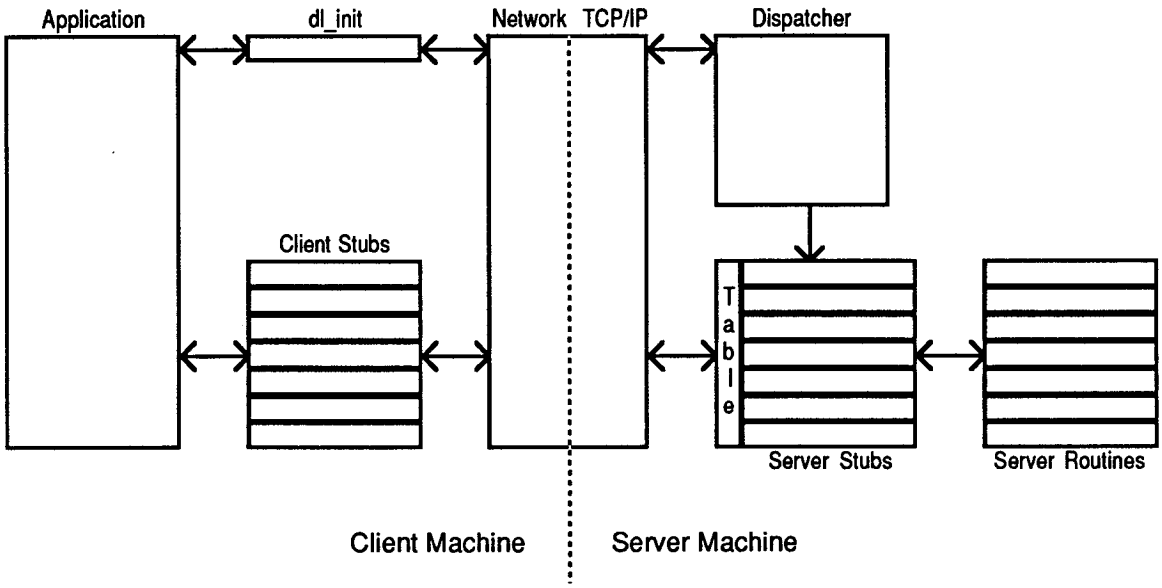

Figure 2. Basic Architecture of Distributed Library

The dispatcher in the above diagram is a daemon process which accepts dlib connections, authenticates users and creates a new process to execute the client's remote calls. It is this new process which becomes the remote environment for the client. The new process maintains remote state which is persistent from call to call. The basic dlib includes a mechanism for allocating memory in this remote environment and the ability to synchronize remote and local buffers.

Once the connection between the client and server is established, the client may proceed to make calls to server routines via the client stub routines. The server maintains a routine switch table corresponding to the remote routines it is capable of servicing and references to the appropriate stub routines. It is the server stub routines which make the actual calls to the server routines.

Client stubs are maintained in libraries which are linked with the application to form the executable code. Server stubs and server routines are also maintained in libraries. A server routine switch table is

maintained to associate information received in a network message with a particular server stub and routine. This protocol is described in the Procedure Execution and Data Representation section below.

The major issues to be faced by a distributed processing system in a heterogeneous network are remote and local process rendezvous, procedure execution and data representation, remote memory management and synchronization, and error handling.

4. Remote and Local Process Rendezvous

The network facilities provided in the Berkeley Standard Distributions of Unix [3,5,6], also known as sockets, are useful in the implementation of the rendezvous portion of the dlib protocol. The system call "rexec" [3] returns a stream to a remote command. In this instance the remote command is the dlib back-end. Rexec handles user authentication and the access to stdin, stdout, and stderr streams for the remote process. A mechanism for signaling the remote process is also provided. In addition to the connection used by rexec, a new connection is made to the remote process to handle the dlib protocol. The dispatcher of the dlib model is handled by rexecd [3].

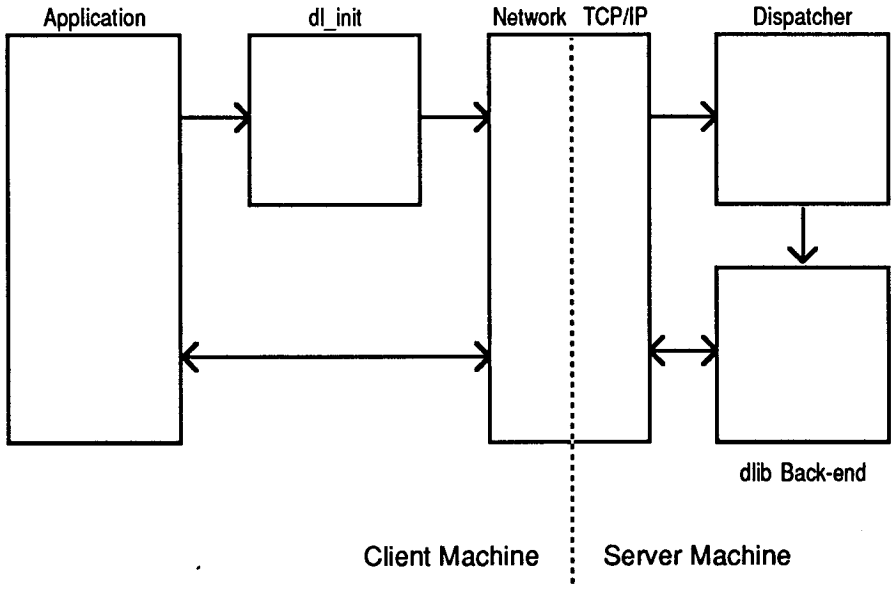

Figure 3. Rendezvous

With the successful completion of the rendezvous, the client has three open connections to the server process:

i. a connection for stdin and stdout

ii. a connection for stderr and signals for the server process

iii. a connection for the dlib protocol

The client process contains client application code and a dlib front-end consisting of local state information and a library of client stubs. The server process contains the dlib back-end which maintains server state information, a library of server stubs which call dlib routines, and a table which matches client calls and dlib routines.

5. Procedure Execution and Data Representation

Once the rendezvous is complete, the client makes remote calls and the server executes them in a process similar to the simple RPC diagrammed in Figure 1. The main difference is that the server is a full (heavyweight) Unix process which maintains remote state until the connection is terminated. The dlib server process maintains a routine switch table which contains the names and locations of the stub routines for the server routines that the server process is capable of executing. The dlib server process is the environment in which multiple successive remote calls may be executed.

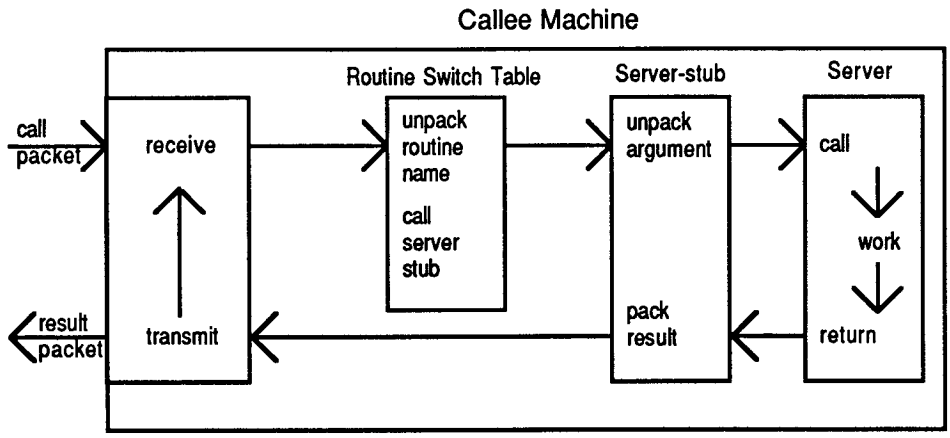

Figure 4. Components and Interactions of a Distributed Library Server Process

The client application makes a dlib call by calling the client stub. The client stub is created by the client stub generator utility. This utility creates the client stub routine from the declaration portion of the remote procedure. The client stub generator extracts the argument type information from the call declaration and develops a string format descriptor for the arguments in the style of the format descriptors of sprintf and sscanf [1, 3]. Sprintf is then used to pack arguments into a string. This string, along with the routine name, is packaged and sent to the server using a TCP/IP network message. The client uses sscanf to unpack a return value from the string which is returned from the server.

The dlib back-end is the main part of the server which receives network messages from the client. The dlib back-end unpacks the name of the routine to be called from the network message. A routine switch table with

the server stub routines indexed by the routine name is searched for a matching name and, if found, the associated server stub routine is called. The argument string from the network message is passed as an argument to the server stub.

The server stub is created using the server stub generator. Similar to the client stub generator, the server stub generator uses the routine declaration to create a server stub. The resulting server stub expects the argument string to correspond to the string format determined by the server stub generator's extraction of the argument types from the call declaration. Variables are declared to contain the arguments. These variables are assigned values using the argument string and string format descriptor as arguments to sscanf.

Once the server stub has extracted the arguments from the argument string, the actual server routine is called using the extracted arguments. The return value is translated to a string and sent in a TCP/IP network message back to the client. The client stub completes the call by extracting the return value from the network message and returning it to the application.

Finding a simple method for solving the problem of network data representation is the motivation for converting the arguments into a string for transmission across the network and extracting them from the string on reception. Since the dlib protocol was intended for use on a heterogeneous network, it can be expected that there will be a variety of data representation schemes employed within the dlib protocol's domain. Using a network canonical form for data representation is a well established method for reducing the number of different translation methods required. The choice of character strings as the canonical form was made since Unix, as defined by the System V Interface Definition [1], contains the translation routines sprintf and sscanf. The translation routines are already implemented on the target machines (the NPSN at NASA Ames). This does not preclude the use of dlib on systems other than Unix, but implementation of translation routines with the semantics of sprintf and sscanf would be required. Using strings in this way certainly is not the most efficient data representation format in terms of size or translation speed. However, in the dlib protocol the canonical data representation is used only for transmission of arguments and return values. This tends to entail relatively small amounts of data to be translated. Consequently, the impact of using a relatively inefficient data representation such as strings is limited.

The following is an example of a dlib routine, dl_open, which opens a file on the remote machine:

```
int  dl_open(path, oflag, mode)
        DL_STRING       path;
        int             oflag;
        int             mode;
```

```
{
        int     fdes;
        fdes = open(path, oflag, mode);
        return(fdes);
}
```

The client stub generator produces a stub routine which translates the arguments to dl_open to a string using the string format descriptor "%s %d %d ". This resulting argument string and the routine name are sent to the server using a TCP/IP network message. The call:

dl_open("/tmp/foo", (O_CREAT|O_TRUNC|O_RDWR), 0644)

results in the string "/tmp/foo 102 420 " where (O_CREAT|O_TRUNC|O_RDWR) is equal to 102, and 0644 is equal to 420 in decimal representation (%d for sprintf).

On reception of the TCP/IP network message the dlib back-end locates the server stub for dl_open in the server routine switch table and calls the stub routine with the argument string as an argument. The server stub for dl_open has allocated space for a string and two integers. The string format descriptor, "%s %d %d ", is used with sscanf to extract the argument values from the argument string "/tmp/foo 102 420 ". The server stub calls the server routine dl_open with the extracted arguments, a character pointer to the a string "/tmp/foo", oflag with a value of 102 and mode of value 420. A successful open returns a file descriptor (an integer). This file descriptor is translated into a string by the server stub using "%d " as the string format descriptor. This string is sent back to the client. The application receives an integer value, which is extracted by the client stub, representing the remote file descriptor.

6. Remote Memory Management and Synchronization

The dlib server process provides a base for the maintenance of remote state, but in order to fully utilize this remote environment the ability to allocate memory and manipulate data stored in that memory is required. There are two basic ways to allocate remote memory: remote external declarations and dl_malloc.

Since the server routines are linked in an executable program, external variables may be declared as in any other routine and they may be used as hidden arguments to dlib routines. They may also be manipulated as a side effect of a dlib routine. This exemplifies the persistent nature of the remote environment.

Dl_malloc is an analog to the local memory allocation routine malloc. Malloc returns an address of a block of memory. Dl_malloc returns a memory descriptor of type DL_MEMDES. The memory descriptor on the client machine is an integer which is a reference to a remote memory segment. The memory descriptor is actually an index to an array of

memory locations stored on the server machine. A variable of type DL_MEMDES may be used in dlib routines much like a character pointer is used in the utilization of dynamically allocated local memory.

Routines are provided to synchronize remote memory segments and local memory segments. These are the only routines which transmit data, other than arguments and return values, over the network. Data representation translations are not performed on synchronization operations. Where data translation is necessary, it is performed on either the local or remote machine with a separate translation routine. While this scheme for data representation manipulation is not automatic it allows the user to control data representation as it is appropriate for the application and environment.

7. Error Handling

There are several methods available for handling errors in dlib. The typical Unix method of returning an invalid value, such as -1 or NULL, and setting the global variable errno to the error code is somewhat replicated in dlib. The first memory descriptor (DL_MEMDES number 0) is reserved to hold the address of a buffer which contains the system error message for the last system error made by the server process. This message is available for printing by the client process using pdlerror, a dlib analog for perror.

Other error handling methods are available to the user. Since stdout and stderr streams are available (via rexec), messages can be printed much in the way one uses stdout and stderr in a local procedure. One difference is that an explicit flush of the stream used must be made in order to cause the message to be transferred from the server to the client.

8. Experiences Using Dlib

Initially, dlib was implemented as a special-purpose protocol to be used over HYPERchannel [9]. The network transport mechanism was implemented in user space utilizing raw HYPERchannel devices. This approach was dropped in favor of using a standard transport protocol, TCP/IP, and the socket network utilities of BSD Unix.

The appeal of a combination of the computational power of the supercomputer and the high-performance graphics of workstations has always been weighed against the difficulty of integrating the capabilities of these disparate systems under the control of a single application. One of the purposes of dlib is to be a tool for developing distributed applications for interactive visualization of computational fluid dynamics (CFD).

CFD data typically consists of grid data and flow solution data. The grid data describes the locations in space (x-, y-, and z-coordinate positions) for which flow data has been calculated. Flow solution data typically consists of energy, density, and momentum values. One set of grid and solution data is the result of a steady state flow solution. For a time accurate flow

solutions in which the flow changes over time are calculated, one set of solution data results from each time step calculated.

An enormous amount of raw data can be generated by CFD flow solvers, particularly for time accurate flow solutions. Grid sizes can be as large as a million nodes or more and as many as a thousand time steps can be accumulated. This adds up to a potential data set of tens of gigabytes. While many of the computational requirements of analyzing this large of a data set can be met by a high-performance graphics workstation, economic factors conspire to make the accumulation of these features on a workstation unrealistic.

Using dlib, the graphics workstation can utilize the supercomputer's computational capabilities to:

i. dynamically allocate large amounts of memory with dl_malloc

ii. utilize the supercomputer's high speed, high capacity disk storage using dlib I/O routines

iii. utilize the supercomputer's number crunching capabilities using dlib routines which perform numerical calculations

Adding these capabilities to a graphics system already rich in user-interface and display capabilities produces an extremely powerful tool.

9. Conclusion

Early implementations of the remote procedure model sought to provide a transaction oriented service for distributed processing. While this type of service is useful for the development of many distributed applications, it does not provide for all distributed processing requirements. Distributed Library takes a different approach in its view of the client-server relationship providing a persistent environment for distributed applications.

10. Acknowledgements

The author would like to thank Eric Raible for the informative discussions on the use of yacc and lex in automatic stub generation. The author would also like to especially thank E. Lisette Gerald for reviewing early versions of this manuscript and suggesting numerous improvements to the final presentation.

11. References

[1] AT&T, *System V Interface Definition*, Volume 1, pp. 199-203, 1986.

[2] A.D. Birrell and B.J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems* 2:1, pp. 39-59, January, 1984.

[3] Computer Systems Research Group, *Unix Programmer's Reference Manual, 4.3 Berkeley Software Distribution*, Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, April 1986.

[4] T.H. Dineen, P.J. Leach, N.W. Mishkin, J.N. Pato, and G.L. Wyatt, "The Network Computing Architecture and System: An Environment for Developing Distributed Applications," *Proceedings of Summer Usenix*, pp. 385-398, June 1987.

[5] S.J. Leffler, R.S. Fabry, W.N. Joy, and P. Lapsley, "An Advanced 4.3BSD Interprocess Communication Tutorial," *Unix Programmer's Manual Supplementary Documents Volume 1, 4.3 Berkeley Software Distribution*, PS1:8-1-41, April 1986.

[6] S. Sechrest, "An Introductory 4.3BSD Interprocess Communication Tutorial," *Unix Programmer's Manual Supplementary Documents Volume 1, 4.3 Berkeley Software Distribution*, pp. PS1:7-1-25, April 1986.

[7] Sun Microsystems, *Request for Comment #1057*, Network Working Group, June 1988.

[8] Xerox Corporation, "Courier: The Remote Procedure Call Protocol," *Xerox System Integration Standard (XSIS) 038112*, December, 1981.

[9] M.J. Yamasaki, "Special Purpose User-Space Network Protocols," *Proceedings of Winter Usenix*, pp. 63-69, February 1988.